ML Assignment 4

Shiva Kumar Dande

NUID: 002702631

Question 1

Data Distribution:

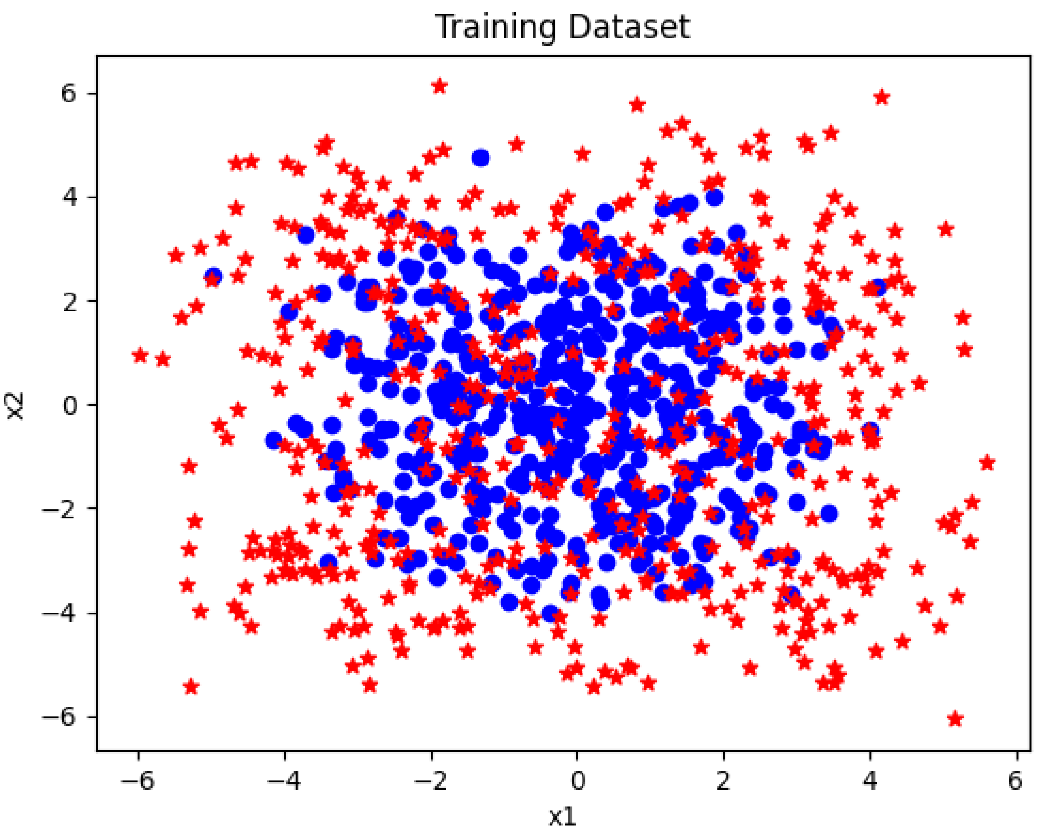
Using the equation given below, 1000 iid samples are generated for training purpose and 10000 iid testing samples are created for final model evaluation.

$$x = rl\,[\,\boldsymbol{\cos\theta}\ \boldsymbol{\sin\theta}\,] + n$$

where $\theta \sim$Uniform $[-\pi, \pi]$ and $n \sim$ Normal $(0, I)$.

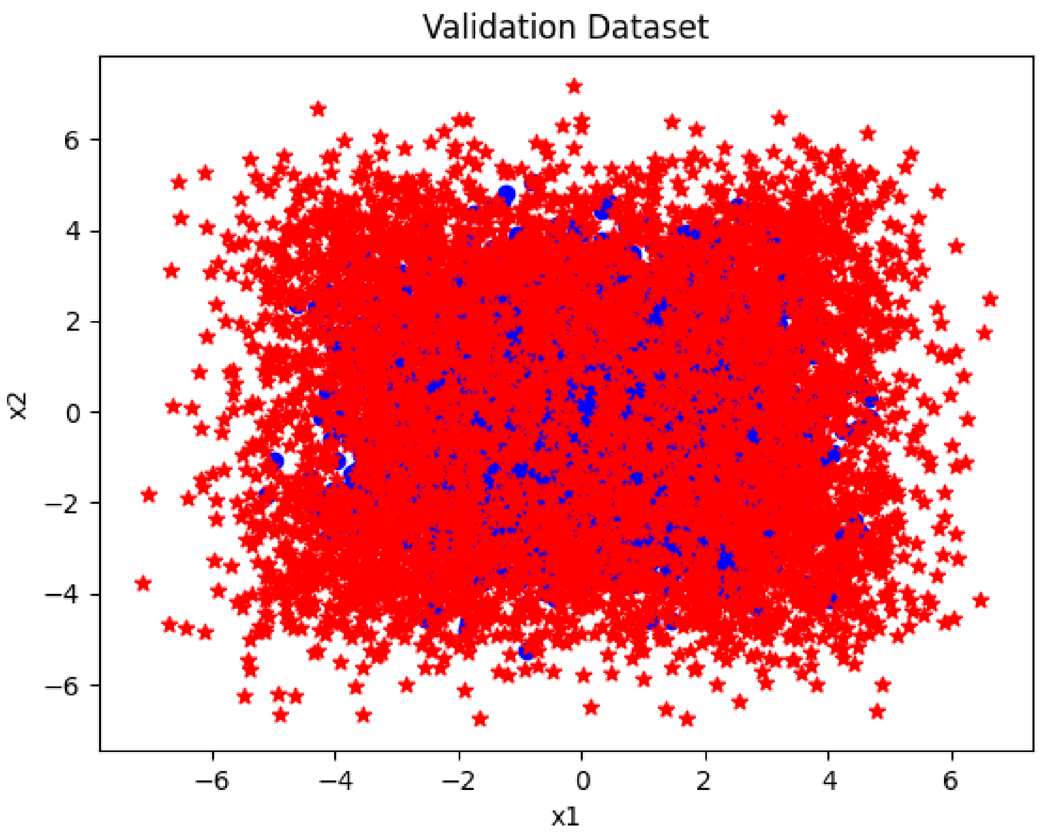
The data samples belong to classes with labels -1 and 1 having uniform priors and each sample is a 2-dimensional vector. The value of r is decided based on the class label of the data sample as - $r-1 = 2$ and $r+1 = 4$.

Training Dataset:



As can be seen, there is a significant overlap between the two classes. The data points are trained using both SVM and MLP classifiers to compare their performance.

Validation Dataset:

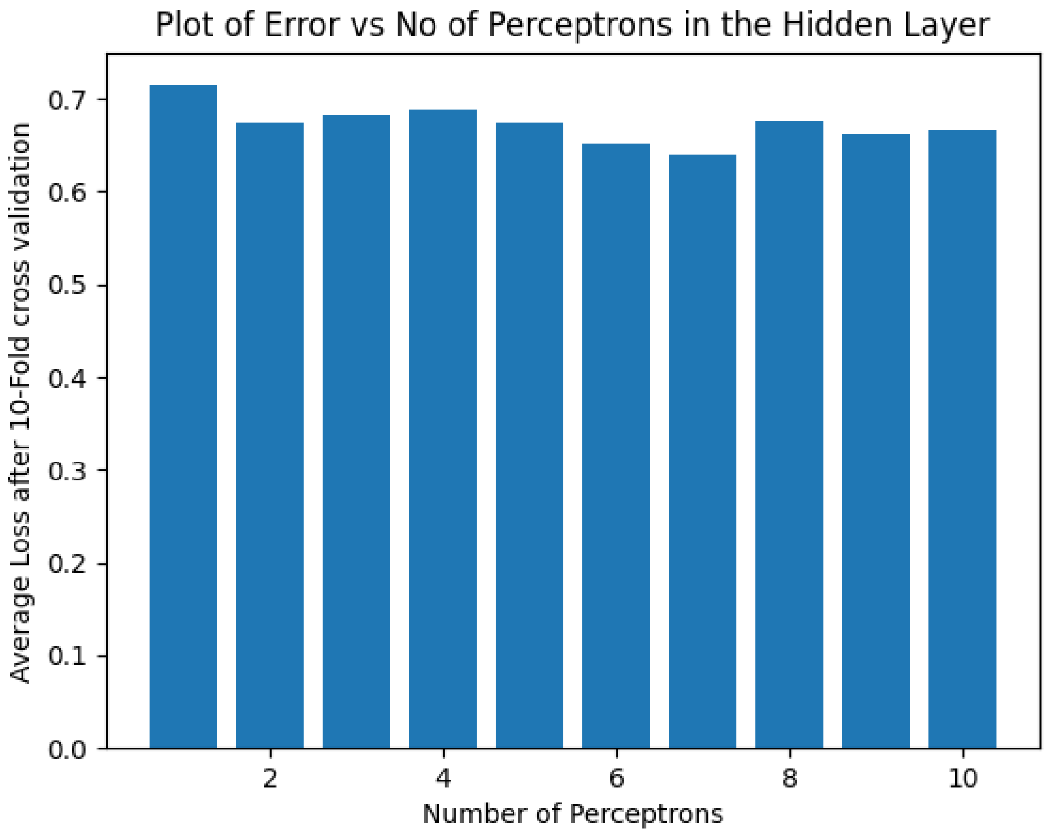


Multi-Layer Perceptron (MLP)

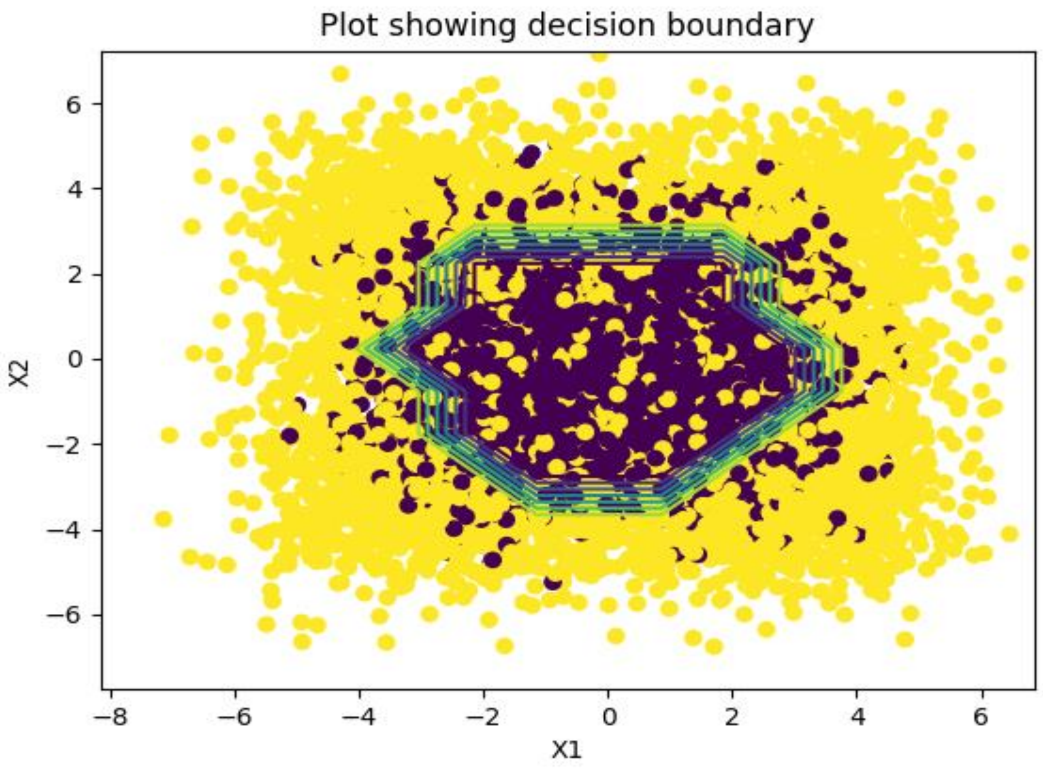
In this case, a single hidden layer Neural Network is employed for binary classification tasks. The class label -1 is substituted with 0 to enable training with PyTorch. The input layer consists of two neurons that represent the dimensions of the feature vector, while the output layer is composed of a single neuron followed by a sigmoid layer, which produces class probabilities. A threshold of 0.5 is applied, with values below it predicted as class 0 and values above it classified as label 1. Given the data distribution and the elliptical boundary between classes, a custom quadratic activation function is utilized in the hidden layer.

The number of perceptrons in the hidden layer is a hyperparameter, and the optimal quantity is determined through 10-fold cross-validation. Binary Cross Entropy is chosen as the cost function to reduce the error, with Stochastic Gradient Descent (SGD) serving as the optimization function. SGD is effective in avoiding local minima due to its inherent randomness in updating parameters after each sample, while the added momentum ensures rapid convergence.

The average validation loss after the cross-validation process is compared for models with varying numbers of neurons, ranging from 1 to 14, as illustrated in the bar graph below.



It is evident that after 7 neurons, the average loss attains a minimum value and begins to plateau. Therefore, 7 neurons are employed in the hidden layer to train the final model. The final model is assessed using 10,000 test samples, with the results presented below.



FINAL LOSS IS 0.49982333183288574, ACCURACY IS 0.4912
Confusion Matrix:

$$\begin{bmatrix} 3925 & 987 \\ 1352 & 3736 \end{bmatrix}$$

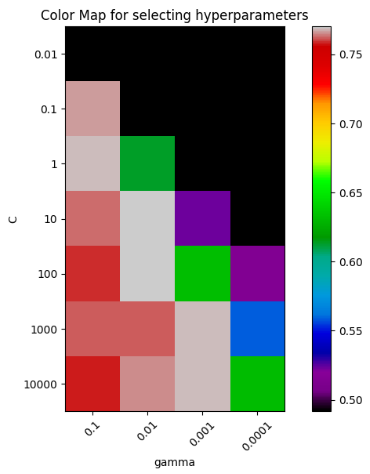
Error: 0.2339

Support Vector Machine (SVM):

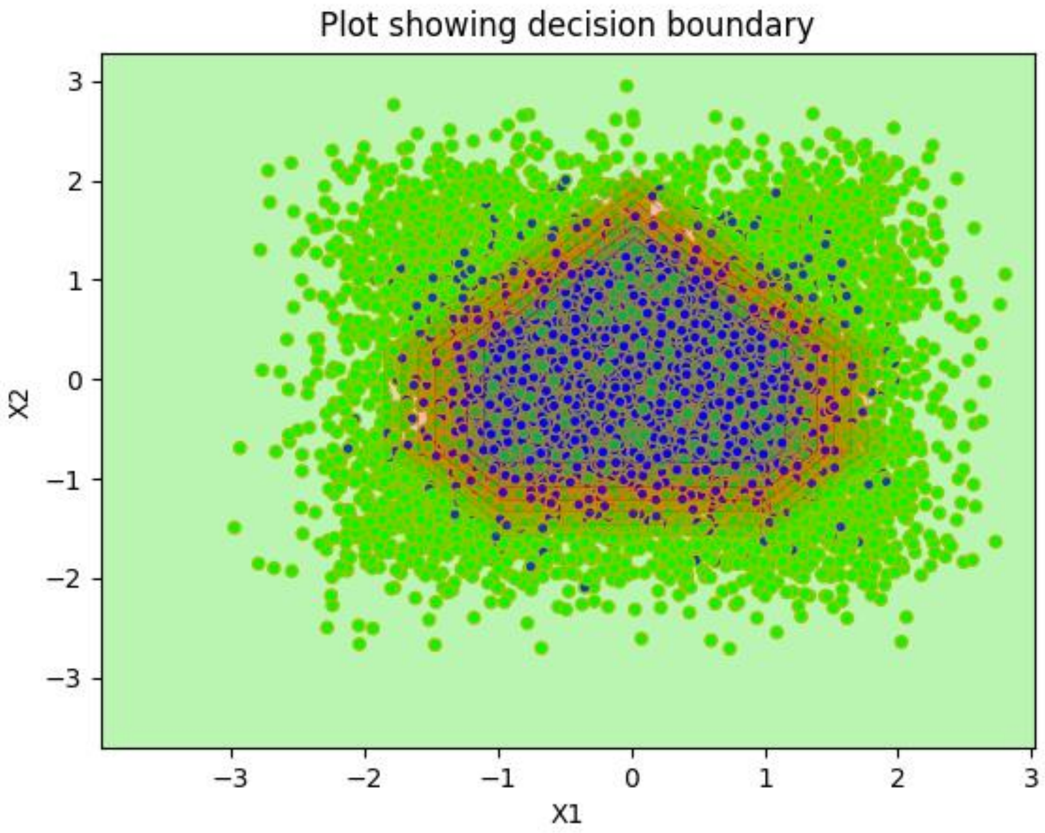
A Gaussian Kernel (Radial Basis Function) based SVM classifier is employed to obtain a classification boundary between the two classes. Given the data distribution, it is clear that the decision boundary will be non-linear. Thus, the kernel trick is used, which transforms the data points, making them linearly separable by a hyperplane.

Two hyperparameters, C (box constraint) and γ (width of the Gaussian kernel), are identified using 10-fold cross-validation. The value of C controls the trade-off between error for misclassified samples and maximizing the margin between the two classes in the optimization function.

The Scikit-Learn library's SVC() implementation of SVM is used to build the model. The GridSearchCV() function from Scikit-Learn assists in looping through a set of hyperparameters and evaluating the SVM model with each hyperparameter combination. The combination with the highest average accuracy score after cross-validation is selected. The hyperparameter values were varied in multiples of 10, and the results were visualized using the color map below.



A maximum accuracy value of 76.44% is achieved for the combination of C = 100 and γ = 0.01. This can also be observed in the color map, aided by the scale on the right. Using these hyperparameters, the final model is trained and tested on a new set of 10,000 samples, with the results shown below.



Blue points outside and green points inside the decision boundary represent misclassified points.

The features X1 and X2 are standardized values obtained using the transform() function from the Sklearn library. SVM, being a distance-based algorithm, requires all the features to be on a uniform scale for effective sample classification.

Confusion Matrix:

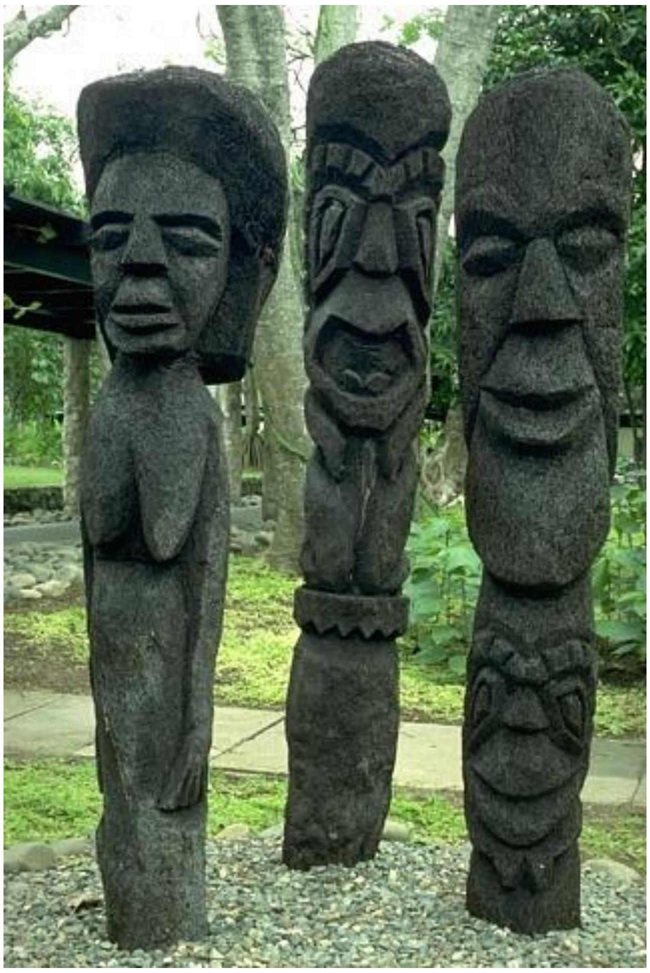
```
[[4190  722]
 [1634 3454]]
```

The accuracy of the model is 76.44 %
Minimum Probability of Error = (100 – 76.44) % = 23.56%

Therefore, both the MLP and SVM classifiers yield comparable performance. The Gaussian Kernel transformation in SVM and the quadratic activation function in MLP contributed to the separation of the highly overlapping data samples.

Question 2:

This problem involves segmenting an image using Gaussian Mixture Model (GMM) based clustering. The selected image is shown below.



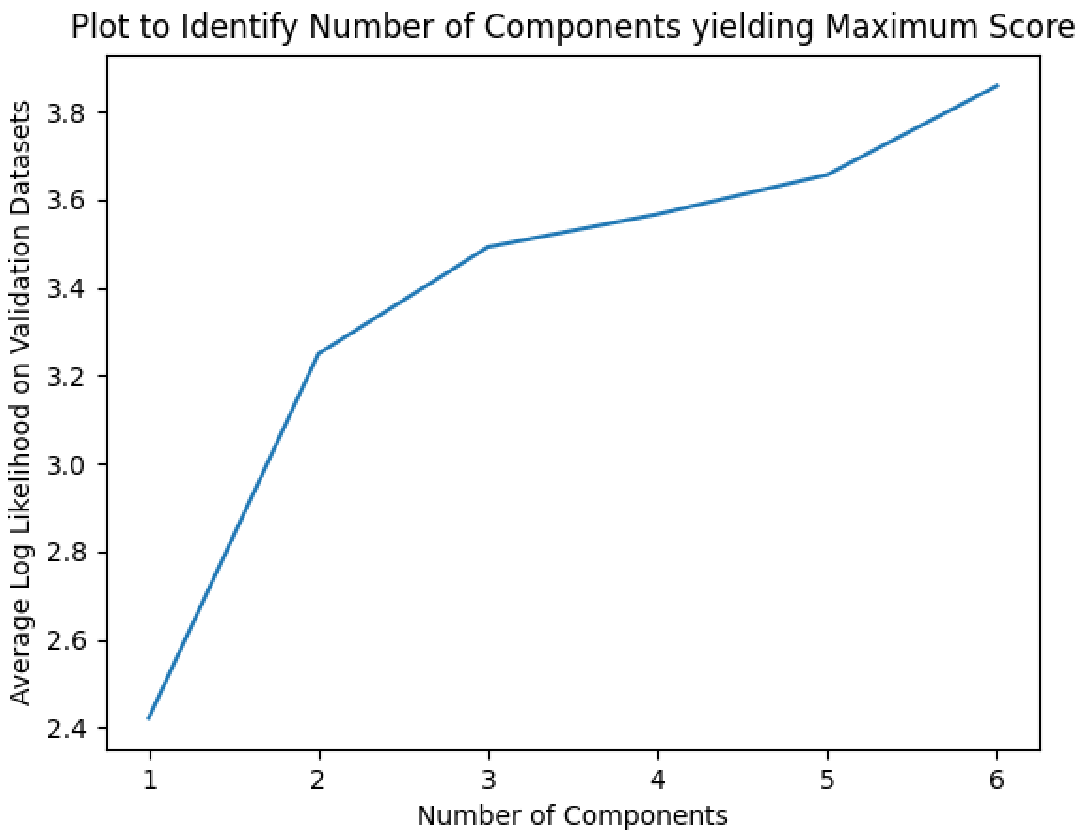
This image contains 154,401 pixels (481 rows and 321 columns), with 3 channels representing the intensity of red, green, and blue colors. To reduce computational load, the image is downsampled to 60% of its original size using OpenCV functions.

Next, the following features are extracted from each pixel of the image to train the model:

1 - Row index of the pixel
2 - Column index of the pixel
3 - Red color intensity
4 - Green color intensity
5 - Blue color intensity

These features are normalized to bring their values to a uniform scale between 0 and 1. The MinMaxScaler() class and fit_transform() function from the Scikit Learn library are used to subtract the minimum value from a feature and divide it by the range of feature values. This ensures all values fit into the 5-dimensional hypercube.

Using 10-fold cross-validation, multiple GMM models are trained with varying numbers of components to identify the best case. The average log-likelihood score after cross-validation is computed for each model with different components, selecting the one with the highest score. The GaussianMixture() and KFold() classes of Scikit Learn provide functions to build the GMM model and split the dataset into training and validation folds, respectively.



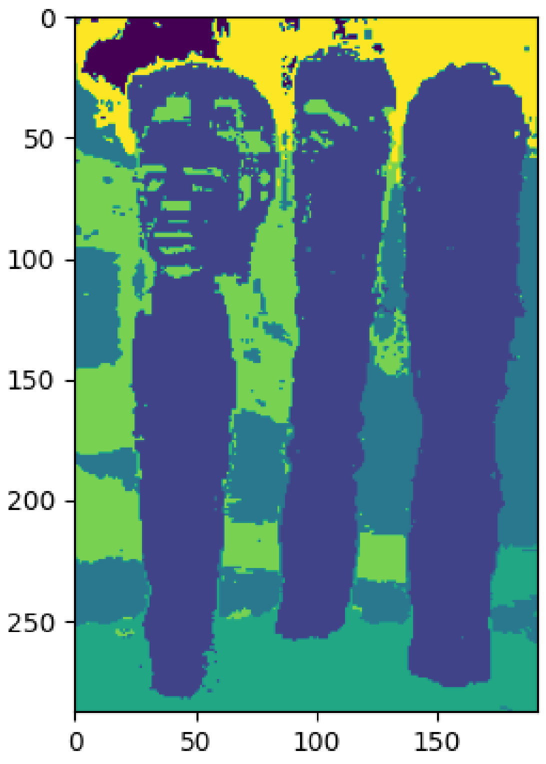
It has been observed that the maximum average log-likelihood score is obtained for 6 Gaussian components. Therefore, the final GMM model is trained with 6 components, which produces the probability of each pixel belonging to different colour clusters. The Maximum A Posteriori (MAP) classification rule, as shown below, is used to compute the posterior probabilities and assign pixels to a specific cluster.

Cluster(L|X) = argmax P(L) * PDF(X|L) where L is cluster number and X is feature vector
L ε {1,..6}

P(L) is the model weights (priors) computed for each gaussian component.

PDF(X|L) is the multivariate probability distribution for the feature vector X evaluated with the help of mean and covariance matrices computed by the model for that cluster.

Comparison of Segmented Image with Original:



The segmentation can successfully distinguish the three rocks, sky, and ground with different colour intensities.

Appendix:
Question1:
Data Generation:

```
import numpy as np
import pandas as pd
from numpy.linalg import inv
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

No_features = 2

def dataset_creation():
    Ntrain = [1000]
    for Nsamples in Ntrain:
        data, labels = generateData(Nsamples)
#          stacking the data (3 X 100) and labels (1 X 100) to create a data
set of size 100 X 4
        data_set = pd.DataFrame(np.hstack((data, labels)))
        plot3(data, labels, 'T')
#          stroing the egenrated data set in a .csv file
        filename = 'training_dataset' + str(Nsamples) + '.csv'
        data_set.to_csv(filename, index = False)

#      creating validation data sets
    NValidation = 10000
    data, labels = generateData(NValidation)
    plot3(data, labels, 'V')
    data_set = pd.DataFrame(np.hstack((data, labels)))
    filename = 'validation_dataset' + str(NValidation) +'.csv'
    data_set.to_csv(filename, index = False)

def generateData(N):
    priors = [0.5, 0.5] # priors should be a row vector
    labels = np.zeros((1, N))
    labels = (np.random.rand(N) >= priors[1]).astype(int)
    labels = np.array([int(-1) if (t == 0) else int(1) for t in labels])

    X = np.zeros(shape = [N, No_features])
    for i in range(N):
        if labels[i] == 1:
            X[i, 0] = 4 * np.cos(np.random.uniform(-np.pi, np.pi))
            X[i, 1] = 4 * np.sin(np.random.uniform(-np.pi, np.pi))
        elif labels[i] == -1:
            X[i, 0] = 2 * np.cos(np.random.uniform(-np.pi, np.pi))
            X[i, 1] = 2 * np.sin(np.random.uniform(-np.pi, np.pi))
        X[i, :] += np.random.multivariate_normal([0, 0], np.eye(2))
    labels = np.reshape(labels, (N,1))
    return X, labels
```

```
def plot3(data, labels, dtype):
#   from matplotlib import pyplot
#   import pylab
#   from mpl_toolkits.mplot3d import Axes3D
    # fig = plt.figure()
    # ax = fig.add_subplot(111, projection='3d')
    plt.scatter(data[(labels.ravel() == -1), 0], data[(labels.ravel() == -1),
1], marker = 'o', color='b')
    plt.scatter(data[(labels.ravel() == 1),0], data[(labels.ravel() == 1), 1],
marker = '*', color='r')
    plt.xlabel("x1")
    plt.ylabel("x2")
    if (dtype == 'T'):
        plt.title('Training Dataset')
    else:
        plt.title('Validation Dataset')
    plt.show()

def predict(data, e):
    return np.matmul(e[:,0], pow(data,3)) + np.matmul(e[:,1], pow(data,2)) +
np.matmul(e[:,2], pow(data,1)) + np.matmul(e[:,3], np.ones((2,1)))

dataset_creation()
```

MLP:

```
# Load the necessary Python libraries
# such as NumPy, Pandas, Matplotlib, Scikit-learn, and PyTorch.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

# Define the MLP structure with two layers, a hidden layer, and an output
layer.
# The number of perceptrons in the hidden layer should be determined by cross-
validation.
# You can experiment with different smooth-ramp activation functions such as
# ISRU, Smooth-ReLU, ELU, etc., to see which one works best for your problem.
class MLP(nn.Module):
    # def __init__(self, *args, **kwargs) -> None:
```

```python
    #     super().__init__(*args, **kwargs)
    def __init__(self, input_dim, hidden_dim, output_dim) -> None:
        super().__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        self.activationMethod = nn.ELU(alpha = 1.0)
        self.layer2 = nn.Linear(hidden_dim, output_dim)
        self.Sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.layer1(x)
        x = self.activationMethod(x)
        x = self.layer2(x)
        x = self.Sigmoid(x)
        return x

# Train your MLP using the training set.
# During training, you will need to feed the input data forward
# through the MLP to compute the output, then backpropagate the
# error to update the weights. Repeat this process until the MLP converges,
# or until a predetermined stopping criterion is met.
def train(model, loss_fn, optimizer, train_loader):
    # Set the model to train mode
    model.train()
    # Iterate over the batches of training data
    for batch in train_loader:
        # Zero the gradients for this batch
        # Compute the model's predictions for this batch
        inputs, labels = batch
        inputs = inputs.float()
        outputs = model(inputs)
        # Convert the labels tensor to Long type
        # labels = labels.long()
        # print(outputs)
        outputs = outputs.squeeze(1)
        # print(outputs.shape, labels)
        # Compute the loss between the predictions and the ground-truth labels
        loss = loss_fn(outputs.float(), labels.float())
        # Backpropagate the loss and update the weights
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return outputs, loss

# Evaluate the model's performance on the validation set after each epoch
def evaluate(model, loss_fn, test_loader):
    model.eval()
    with torch.no_grad():
        total_loss = 0
        total_correct = 0
        total_samples = 0
        for batch in test_loader:
            inputs, labels = batch
            outputs = model(inputs.float())
            # Convert the labels tensor to Long type
            # labels = labels.long()
            loss = loss_fn(outputs.squeeze(1).float(), labels.float())
            total_loss += loss.item() * inputs.size(0)
            total_correct += (outputs.argmax(dim=1) == labels).sum().item()
            total_samples += inputs.size(0)
        val_loss = total_loss / total_samples
        val_acc = total_correct / total_samples
    return val_loss, val_acc, outputs

# Randomly initialize weights
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

def plotBarGraph(average_loss):
    # Plot a bar graph to decide optimal number of perceptrons
    plt.bar(np.linspace(1,10,10), average_loss)
    plt.xlabel("Number of Perceptrons")
    plt.ylabel("Average Loss after 10-Fold cross validation")
    plt.title("Plot of Error vs No of Perceptrons in the Hidden Layer")
    plt.show()

# Load your dataset into a Pandas dataframe or NumPy array.
data = pd.read_csv('training_dataset1000.csv')
input_features = data.iloc[:, : -1].to_numpy()
labels = data.iloc[:, -1].to_numpy()
labels = np.array([0.0 if i == -1 else 1.0 for i in labels])

#X_train, X_test, y_train, y_test = train_test_split(data.iloc[:, : -
1].to_numpy(), data.iloc[:, -1].to_numpy(), test_size=0.2, random_state=42)
batch_size = 32
num_folds = 10
kf = KFold(n_splits=num_folds)

minValidationLoss = float("inf") # Assining a large value to minValidationLoss
(+ infinity)
maxValidationAccuracy = float("-inf") # Assining a large value to
maxValidationAccuracy (- infinity)
optimal_no_of_Perceptrons = 0
average_loss = []
```

```
for perceptrons in range(1, 11):
    fold_loss = []
    fold_accuracy = []
    print('-----------------------------------------------------------------
--')
    print(f"Number of Neurons / Perceptrons in the Hidden Layer :
{perceptrons} ")
    print('-----------------------------------------------------------------
--')
    model = MLP(input_features.shape[1], perceptrons, 1)
    model.apply(init_weights)

    # Define the loss function, which in this case, is cross-entropy loss.
    # The loss function will be used to evaluate how well your MLP is
performing.
    loss_fn = nn.BCEWithLogitsLoss()

    # Define the optimizer, which will be used to update the weights of your
MLP.
    # The most commonly used optimizer is stochastic gradient descent (SGD),
    # which updates the weights after each batch of training examples.
    import torch.optim as optim
    learning_rate = 0.001
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum =
0.9)

    for fold, (train_index, test_index) in
enumerate(kf.split(input_features)):
        # print(f'Fold {fold + 1}')
        # Split data into train and test sets for the current fold
        X_train, X_test = input_features[train_index],
input_features[test_index]
        y_train, y_test = labels[train_index], labels[test_index]

        # Create PyTorch data loaders for the train and test sets
        train_dataset = TensorDataset(torch.from_numpy(X_train),
torch.from_numpy(y_train))
        train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

        test_dataset = TensorDataset(torch.from_numpy(X_test),
torch.from_numpy(y_test))
        test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

        for epoch in range(50):
            # Forward pass
            outputs, loss = train(model, loss_fn, optimizer, train_loader)
            # Evaluate your MLP using the testing set.
            # This will give you an idea of how well your MLP generalizes to
new data.
            validationLoss, validationAccuracy, predictions = evaluate(model,
loss_fn, test_loader)

            # Print the epoch number, training loss, and validation loss and
accuracy
            # if epoch % 200 == 0:
        # print(f'Epoch {epoch}: TRAINING LOSS = {loss:.4f}, VALIDATION LOSS =
{validationLoss:.4f}, VALIDATION ACCURACY = {validationAccuracy:.4f}')

        # Evaluate the model's performance on the current fold and save the
error
        loss, accuracy, predictions  = evaluate(model, loss_fn, test_loader)
        fold_loss.append(loss)
        fold_accuracy.append(accuracy)

     # Calculate the mean error across all folds for the current number of
perceptrons
    mean_error = np.mean(fold_loss)
    mean_accuracy = np.mean(fold_accuracy)
    print(f'Mean error for {perceptrons} perceptrons: {mean_error:.4f}')
    average_loss.append(mean_error)

    if (minValidationLoss > mean_error):
        minValidationLoss = min(mean_error, minValidationLoss)
        maxValidationAccuracy = max(mean_accuracy, maxValidationAccuracy)
        optimal_no_of_Perceptrons = perceptrons

print(f'Optimal number of Perceptrons = {optimal_no_of_Perceptrons} :
VALIDATION LOSS = {minValidationLoss:.4f}, VALIDATION ACCURACY =
{maxValidationAccuracy:.4f}')

# Finally, visualize the results using Matplotlib or other visualization tools
# to better understand the performance of your MLP.
plotBarGraph(average_loss)

# Experiment with different parameters,
# such as the number of perceptrons in the hidden layer,
# the learning rate of the optimizer, the number of training epochs, etc.,
# to see how they affect the performance of your MLP.

# Load your dataset into a Pandas dataframe or NumPy array.
data = pd.read_csv('training_dataset1000.csv')
X_train = data.iloc[:, :-1].to_numpy()
y_train = data.iloc[:, -1].to_numpy()
y_train = np.array([0.0 if i == -1 else 1.0 for i in y_train])

# Load your dataset into a Pandas dataframe or NumPy array.
data = pd.read_csv('validation_dataset10000.csv')
X_test = data.iloc[:, :-1].to_numpy()
y_test = data.iloc[:, -1].to_numpy()
y_test = np.array([0.0 if i == -1 else 1.0 for i in y_test])

model = MLP(input_features.shape[1], optimal_no_of_Perceptrons, 1)
model.apply(init_weights)
# Compute minimum binary cross entropy loss
loss_fn = nn.BCELoss()
learning_rate = 0.001
optimizer = optim.SGD(model.parameters(), lr=learning_rate,  momentum = 0.9)

# Create PyTorch data loaders for the train and test sets
train_dataset = TensorDataset(torch.from_numpy(X_train),
torch.from_numpy(y_train))
train_loader = DataLoader(train_dataset, shuffle=True)
batchSize = X_test.shape[0]
test_dataset = TensorDataset(torch.from_numpy(X_test),
torch.from_numpy(y_test))
test_loader = DataLoader(test_dataset, batch_size = batchSize, shuffle=False)

for epoch in range(250):
    print(epoch)
    # Forward pass
    outputs, loss = train(model, loss_fn, optimizer, train_loader)

    # Evaluate your MLP using the testing set.
    # This will give you an idea of how well your MLP generalizes to new data.
    #validationLoss, validationAccuracy, predictions = evaluate(model,
loss_fn, test_loader)

y_pred = torch.zeros(0, dtype=torch.long, device='cpu')
# Evaluate the model's performance on the current fold and save the error
loss, accuracy, predictions = evaluate(model, loss_fn, test_loader)
print(f"FINAL LOSS IS {loss}, ACCURACY IS {accuracy}")
predictions = torch.flatten(predictions)
y_pred = torch.Tensor([0 if value <= 0.5 else 1 for value in
predictions.detach().numpy()])
ConfusionMatrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(ConfusionMatrix)
error = (1 - (ConfusionMatrix[0][0] + ConfusionMatrix[1][1]) /
X_test.shape[0])
print("Error:")
print(error)
```

```python
x1_min, x1_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
x2_min, x2_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
print(x1_max, x1_min)
xx, yy = np.meshgrid(np.arange(x1_min, x1_max), np.arange(x2_min, x2_max))
Z = model(torch.from_numpy(np.c_[xx.ravel(), yy.ravel()]).float())
Z = np.array([0 if value[0] <= 0.5 else 1 for value in
Z.detach().numpy()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.scatter(X_test[:, 0], X_test[:, 1], c = y_test)
plt.contour(xx, yy, Z)
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("Plot showing decision boundary")
plt.show()
```

SVM:

```python
# Load the necessary Python libraries
# such as NumPy, Pandas, Matplotlib, Scikit-learn, and PyTorch.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

# Load your dataset into a Pandas dataframe or NumPy array.
data = pd.read_csv('training_dataset1000.csv')
input_features = data.iloc[:, : -1].values
sc = StandardScaler()
x_train = sc.fit_transform(input_features)
y_train = data.iloc[:, -1].values

# Load your dataset into a Pandas dataframe or NumPy array.
data = pd.read_csv('validation_dataset10000.csv')
input_features = data.iloc[:, : -1].values
x_test = sc.transform(input_features)
y_test = data.iloc[:, -1].values
```

```
# Define range of hyperparameters for cross validation
C = [0.01, 0.1, 1, 10, 100, 1000, 10000]
gamma = [0.1, 0.01, 0.001, 0.0001]
hyperParameters = {"kernel": ["rbf"], "gamma": gamma, "C": C}

#X_train, X_test, y_train, y_test = train_test_split(data.iloc[:, : -
1].to_numpy(), data.iloc[:, -1].to_numpy(), test_size=0.2, random_state=42)
batch_size = 32
num_folds = 10
kf = KFold(n_splits=num_folds)

# Perform cross validation and identify best results
clf = GridSearchCV(SVC(), hyperParameters, cv = kf)
clf.fit(x_train, y_train)

# printing the parameters for which we get the best results
print(clf.best_params_)

# Print average accuracy scores for all combinations
means = clf.cv_results_["mean_test_score"]
stds = clf.cv_results_["std_test_score"]
for mean, std, params in zip(means, stds, clf.cv_results_["params"]):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))

scores = np.array(means).reshape(7, 4)

plt.figure(figsize=(10, 10))
plt.subplots_adjust(left = 0.15, right = 0.6, bottom = 0.3, top = 0.95)
plt.imshow(scores, interpolation='nearest', cmap = plt.cm.nipy_spectral)
plt.xlabel('gamma')
plt.ylabel('C')
plt.title("Color Map for selecting hyperparameters")
plt.xticks(np.arange(len(gamma)), gamma, rotation=45)
plt.yticks(np.arange(len(C)), C)
plt.colorbar()
plt.show()

# Train final SVM classifier
classifier = SVC(C=10, kernel = 'rbf', gamma = 0.1, random_state = 0)
classifier.fit(x_train, y_train)

# Test the model with appropriate evaluation metrics
predictions = classifier.predict(x_test)
print(predictions)
ConfusionMatrix = confusion_matrix(y_test, predictions)
print("Confusion Matrix:")
print(ConfusionMatrix)
print("The accuracy of the model is {}
%".format(str(round(accuracy_score(y_test,predictions),4)*100)))

# Define range of points for identifying classification boundary
x1_min, x1_max = x_test[:, 0].min() - 1, x_test[:, 0].max() + 1
x2_min, x2_max = x_test[:, 1].min() - 1, x_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max), np.arange(x2_min, x2_max))
Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot classification boundary superimposed over testing data
plt.scatter(x_test[:, 0], x_test[:, 1], c = y_test, cmap = "brg", s = 20,
edgecolors = 'y')
plt.contourf(xx, yy, Z, cmap = "brg", alpha = 0.3)
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("Plot showing decision boundary")
plt.show()
```

Question2:
GMM Modle

```
import numpy as np
import sys
import cv2 as cv
from matplotlib import pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
from scipy.stats import multivariate_normal
np.set_printoptions(threshold=sys.maxsize)

# Read the image from the specified path
image = cv.imread('101085.jpg')
print(image.shape)

# Downsample the image with specified scale
scaling_percentage = 60 # percent of original size
w = int(image.shape[1] * (scaling_percentage / 100))
h = int(image.shape[0] * (scaling_percentage / 100))
dim = (w, h)

# Resizing the input image
img = cv.resize(image, dim, interpolation = cv.INTER_AREA)

# Intialize parameters of the image
rows = img.shape[0]
columns = img.shape[1]
```

```
channels = img.shape[2]
pixels = rows * columns
features = 5

# Create raw feature vector
feature_vector = np.zeros(shape = (pixels, features))
pixel = 0
for row in range(rows):
    for col in range(columns):
        feature_vector[pixel, 0] = row
        feature_vector[pixel, 1] = col
        for channel in range(channels):
            feature_vector[pixel, channel + 2] = img[row, col, channel]
        pixel += 1

# Normalize the feature values
sc = MinMaxScaler()
feature_vector = sc.fit_transform(feature_vector)

# Define number of folds and list to store model orders
kf = KFold(n_splits = 10)
avg_log_likelihoods = []
order =[]
num_components = [1, 2, 3, 4, 5, 6]

# Test model orders ranging from 1 to 6
for num in num_components:
    # Initialize a list to store the log-likelihoods for each fold
    log_likelihoods = []
    for train_index, val_index in kf.split(feature_vector):
        # Split the data into training and validation sets
        X_train, X_val = feature_vector[train_index],
feature_vector[val_index]
        # Fit the GMM using the EM algorithm
        gmm = GaussianMixture(n_components=num, init_params = 'random',
max_iter = 3000, tol = 1e-5, n_init = 2)
        gmm.fit(X_train)

        # Calculate the log-likelihood of the validation set
        log_likelihood = gmm.score(X_val)

        # Append the log-likelihood to the list for this fold
        log_likelihoods.append(log_likelihood)

    # Calculate the average log-likelihood across all K folds
    avg_log_likelihood = np.mean(log_likelihoods)
    #print(log_likelihoods)
    print(avg_log_likelihood)
    # Append the average log-likelihood to the list for all GMMs
    avg_log_likelihoods.append(avg_log_likelihood)

print(avg_log_likelihoods)
# Plot Log Likelihood Score with respect to number of components in each model
plt.plot(np.linspace(1, 6, 6), avg_log_likelihoods)
plt.xlabel("Number of Components")
plt.ylabel("Average Log Likelihood on Validation Datasets")
plt.title("Plot to Identify Number of Components yielding Maximum Score")
plt.show()

# Final Model Fitting
best_num_components = num_components[np.argmax(avg_log_likelihoods)]
print(best_num_components)
best_gmm_model = GaussianMixture(n_components = best_num_components,
init_params='random', max_iter = 3000, tol = 1e-8, n_init = 3)
best_gmm_model.fit(feature_vector)

# Compute class posteriors using model weights and conditional probabilties
posteriors = np.zeros((best_num_components, pixels))
for i in range(best_num_components):
    PDF = multivariate_normal.pdf(feature_vector, mean =
best_gmm_model.means_[i,:], cov = best_gmm_model.covariances_[i,:,:])
    posteriors[i, :] = (best_gmm_model.weights_[i] * PDF)

# Decide label for each pixel with maximum posterior value
img_labels = np.argmax(posteriors, axis = 0)

# Plot segmented image
plt.imshow(img_labels.reshape(rows, columns))
plt.show()
```